Part I.

	n=60		n=120		n=240		n=480		n=960		n=1920	
	w/o blocking	w/ blocking	w/o blocking	w/ blocking	w/o blocking	w/ blocking	w/o blocking	w/ blocking	w/o blocking	w/ blocking	w/o blocking	w/ blocking
N=5	0.054288	0.060009	0.449007	0.486934	4.217585	3.587078	34.275040	28.788939	305.168356	268.909622	2746.208830	2292.225813
N=15	0.055411	0.077835	0.460730	0.531462	4.205563	3.819581	34.688820	29.759629	311.223601	239.162847	2594.194411	1930.242948
N=30	0.054537	0.103465	0.472341	0.804925	4.406616	4.280736	34.682845	31.422497	318.498958	242.380305	2616.268239	1907.093519
N=60			0.450942	0.861160	4.194515	4.870346	36.356412	35.747258	325.104287	258.776055	2632.106841	1731.969366
N=120					4.234965	6.865527	36.156255	41.786863	322.413702	292.588291	2597.324594	1807.634982
N=240							34.809631	59.211691	325.396761	362.014504	2519.050696	2072.200977
N=480									322.663788	526.571019	2348.720396	2657.947531
N=960											2081.343301	3972.827956

Figure 1. Performance of algorithms w/ and w/o blocking with different n and N

1. As you increase the value of N, how do your block sizes change?

Increasing the value of N also increases the block size.

2. For small matrices, does the algorithm with blocking perform better?

No. For the test small matrices of size <200, the algorithm without blocking performed better.

3. Why or why not does blocking affect the performance of the multiplication?

Blocking best works on large matrix sizes where it is not possible to load matrix A, and B, and the results into the cache at the same time which causes more cache misses and slows down the performance.

4. For large matrices, how does the value of N affect the performance of the algorithm with blocking?

Choosing larger block sizes or smaller N, makes the performance faster.

5. Why does N influence the amount of optimization achieved by blocking?

The ultimate goal of blocking is to utilize the cache of the CPU. More specifically, to ensure that all data used for the operation will fit into the cache (matrix A, matrix B, and the results C). Choosing N influences the block size used in the operation. Choosing smaller block sizes or chunks for the matrices results to shorter strides which improves spatial locality and hence, better performance. A block size where all the three matrices A, B, and C does not fit into the cache will have more or less the same performance as the naïve 3 nested loop approach.

Part II.
A.

1. Which library is Numpy configured to use?

Numpy is configured to use 'openblas' in google colab.

2. Try the same command on your own machine. Which library is Numpy configured to use on your machine?

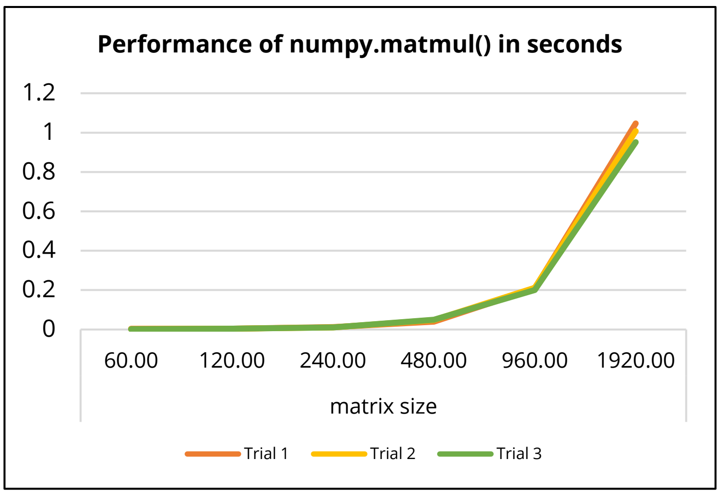
Numpy on my local machine is configured to use 'openblas'.

B.

1. What is the difference in performance of matmul() and the provided matrix multiply code with blocking? Is it a significant difference?

	n=60			n=120			n=240			n=480			n=960			n=1920		
	w/ blocking	matmul()	%	w/ blocking	matmul()	%	w/ blocking	matmul()	%	w/ blocking	matmul()	%	w/ blocking	matmul()	%	w/ blocking	matmul()	%
N=5	0.06081	0.00368	1600	0.45651	0.00375	12100	3.73443	0.01181	31600	29.1236	0.03961	73500	281.956	0.20862	135100	2457.19	1.04658	234700
N=15	0.08201	0.00228	3500	0.52712	0.00352	14900	4.03525	0.0111	36300	29.8539	0.04802	62100	236.538	0.21142	111800	1929.62	1.00778	191400
N=30	0.10572	0.00158	6600	0.62823	0.00404	15500	4.3177	0.0105	41100	31.804	0.04942	64300	242.377	0.20006	121100	1893.45	0.95162	198900
N=60				0.85805	0.00311	27600	5.0826	0.01061	47900	35.8784	0.04215	85100	258.371	0.19165	134800	1939.93	0.95866	202300
N=120							7.23012	0.0114	63400	42.2084	0.04593	91900	295.788	0.17524	168700	2052.11	0.96962	211600
N=240										60.941	0.04134	147400	353.135	0.24659	143200	2310.99	1.00623	229600
N=480													500.598	0.2186	229000	2894.08	0.99572	290600
N=960																4601.5	0.93799	490500

Figure 2. Performance (in secs) of MMM algorithms w/ blocking and matmul() with different n and N



The performance of matmul() is significantly faster than the blocking algorithm. The gap in performance increases as the matrix size increases. Similarly, blocking algorithm performs significantly worse than matmul() as N increases.

2. Can you observe this difference in performance for all sizes of matrices?
Yes.

3. What are the reasons behind the difference in performance?
The big gap in performance between blocking and using matmu() is the latter using BLAS. BLAS contains highly optimized subprograms for linear algebra tasks. Each computer could have different cache hierarchies and sizes among other differences in architecture, and so, BLAS implementation could change depending on the machine's architecture. With this, BLAS implementation could offer increase in performance by ensuring the implementation is optimized for the specific machine's architecture. In the case of matrix-matrix multiplication, BLAS maximizes the usage machine's caches for maximum performance.

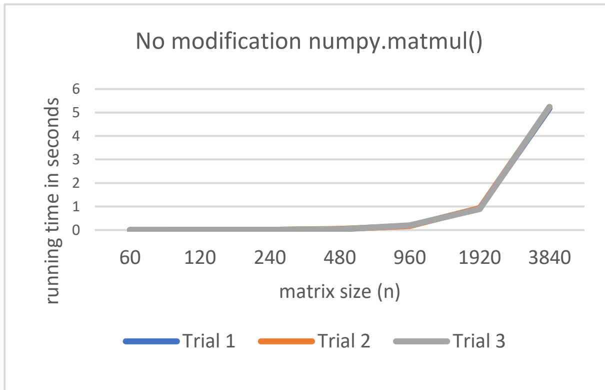
4. How attainable would it be to write your own code that has comparable performance with Numpy's matmul()?
It could be possible but will be extremely challenging. Numpy's matmul() has already been optimized by multiple experts and is already using one of the fastest BLAS library out there.

5. If you were writing an application that performs a lot of linear algebra computations, how should you reorder your code to optimize its performance?

C.

1. What alternative method for performing matrix-
matrix multiplication did you try? Explain how you tried to speed up MMM with this method.

Without modification:



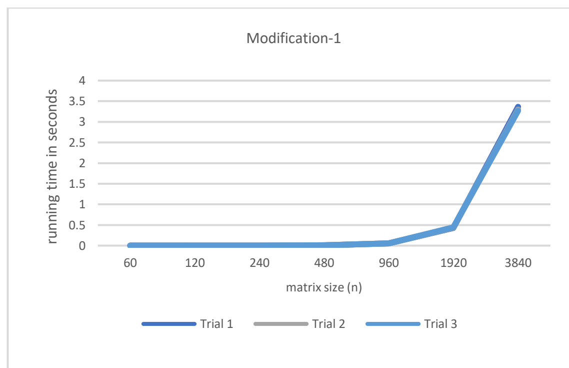
First modification: Converting lists A and B to numpy arrays

```
numpy_A = np.array(A)
numpy_B = np.array(B)

t = time()
c_np1 = np.matmul(A ,B)
runtime = time() - t
print("OG Numpy matmul() completed in %f seconds" % (runtime))

t = time()
c_np2 = np.matmul(numpy_A ,numpy_B)
runtime = time() - t
print("Modified Numpy matmul() completed in %f seconds" % (runtime))

comparison = c_np1 == c_np2
print("Are the two results the same?", comparison.all())
```

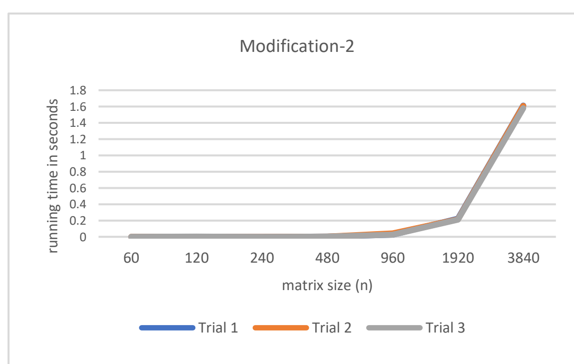


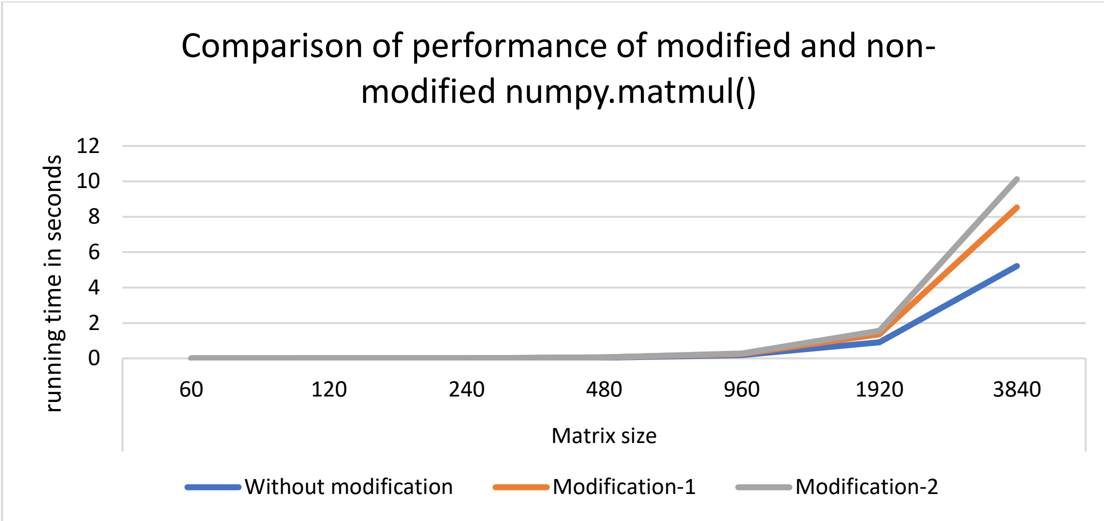
Numpy arrays are better versions of lists in python. Among the many benefits are smaller memory consumption and better performance. Randomly generated arrays were converted to numpy arrays using numpy.array() function. The results for non-modified and modified implementation are compared to check if both results are the same which they are for all n.

Second modification: Converting lists A and B to numpy arrays and changing precision to float32 (original is float64). Numpy arrays generated from the original random 2D lists are converted to float32 using numpy.astype() function.

```
#Second modification
numpy_A_32 = numpy_A.astype('float32')
numpy_B_32 = numpy_B.astype('float32')

t = time()
c_np3 = np.matmul(numpy_A_32 ,numpy_B_32)
runtime = time() - t
print("Modified-2(change floating pt) Numpy matmul() completed in %f seconds" % (runtin
```





2. In your experiments, were you able to speed up matrix-matrix multiplication compared to numpy.matmul()? If yes, did this speed-up apply for all sizes of matrices?

Yes. From the charts of the performances of the modifications in increasing matrices sizes, modification-1 and modification-2 has consistently showed better results than numpy.matmul() without modification in all matrix sizes.